"Express Mail" Mailing Label No. **EL960828258US**

PATENT APPLICATION
ATTORNEY DOCKET NO. SUN-P9699-MEG

METHOD AND APPARATUS FOR DYNAMICALLY ADJUSTING THE AGGRESSIVENESS OF AN EXECUTE-AHEAD PROCESSOR

Inventors: Paul Caprioli and Sherman H. Yip

Related Application

[0001] The subject matter of this application is also related to the subject matter in a co-pending non-provisional application by Shailender Chaudhry and Marc Tremblay, entitled, "Selectively Deferring the Execution of Instructions with Unresolved Data Dependencies as They Are Issued in Program Order," having serial number TO BE ASSIGNED, and filing date TO BE ASSIGNED (Attorney Docket No. SUN04-0182-MEG).

BACKGROUND

Field of the Invention

[0002] The present invention relates to techniques for improving the performance of computer systems. More specifically, the present invention relates

to a method and an apparatus for speeding up program execution by dynamically adjusting the aggressiveness of an execute-ahead processor, wherein the execute-ahead processor selectively defers execution of instructions with unresolved data-dependencies as they are issued for execution in program order.

Related Art

[0003] Advances in semiconductor fabrication technology have given rise to dramatic increases in microprocessor clock speeds. This increase in microprocessor clock speeds has not been matched by a corresponding increase in memory access speeds. Hence, the disparity between microprocessor clock speeds and memory access speeds continues to grow, and is beginning to create significant performance problems. Execution profiles for fast microprocessor systems show that a large fraction of execution time is spent not within the microprocessor core, but within memory structures outside of the microprocessor core. This means that the microprocessor systems spend a large fraction of time waiting for memory references to complete instead of performing computational operations.

[0004] Efficient caching schemes can help reduce the number of memory accesses that are performed. However, when a memory reference, such as a load operation generates a cache miss, the subsequent access to level-two (L2) cache or memory can require dozens or hundreds of clock cycles to complete, during which time the processor is typically idle, performing no useful work.

[0005] A number of techniques are presently used (or have been proposed) to hide this cache-miss latency. Some processors support out-of-order execution, in which instructions are kept in an issue queue, and are issued "out-of-order" when operands become available. Unfortunately, existing out-of-order designs have a hardware complexity that grows quadratically with the size of the issue

queue. Practically speaking, this constraint limits the number of entries in the issue queue to one or two hundred, which is not sufficient to hide memory latencies as processors continue to get faster. Moreover, constraints on the number of physical registers that are available for register renaming purposes during out-of-order execution also limits the effective size of the issue queue.

[0006] Some designers have proposed a scout-ahead execution mode, wherein instructions are speculatively executed to prefetch future loads, but wherein results are not committed to the architectural state of the processor. For example, see U.S. Patent No. 6,415,356, entitled "Method and Apparatus for Using an Assist Processor to Pre-Fetch Data Values for a Primary Processor," by inventors Shailender Chaudhry and Marc Tremblay. This solution to the latency problem eliminates the complexity of the issue queue and the rename unit, and also achieves memory-level parallelism. However, it suffers from the disadvantage of having to re-compute any computational operations that are performed while in scout-ahead mode.

[0007] Hence, what is needed is a method and an apparatus for hiding memory latency without the above-described drawbacks of existing processor designs.

SUMMARY

[0008] One embodiment of the present invention provides an execute-ahead processor system that solves some of the above-described problems. If a data-dependent stall condition is encountered during program execution, the system enters an execute-ahead mode, wherein instructions that cannot be executed because of the unresolved data dependency are deferred, and other non-deferred instructions are executed in program order. If a non-data-dependent stall condition is encountered during execute-ahead mode, the system enters a scout

mode, wherein instructions are speculatively executed to prefetch future loads, but results are not committed to the architectural state of the execute-ahead processor. On the other hand, if an unresolved data dependency is resolved during the execute-ahead mode, the system enters a deferred mode and executes deferred instructions. During this deferred mode, if some instructions are deferred again, the system determines whether to resume execution in the execute-ahead mode. If it determines to do so, the system resumes execution in the execute-ahead mode, and otherwise resumes execution in a non-aggressive mode.

[0009] In a variation on this embodiment, resuming execution in the non-aggressive execution mode involves remaining in the deferred mode until all deferred instructions are executed and the system returns to a normal execution mode.

[0010] In another variation on this embodiment, resuming execution in the non-aggressive mode involves resuming execution in a non-aggressive execute-ahead mode, wherein if a non-data-dependent stall condition is encountered, the execute-ahead processor does not enter the scout mode, but instead waits for the non-data-dependent stall condition to be resolved, or for an unresolved data dependency to return, before proceeding.

[0011] In a variation on this embodiment, while entering the execute-ahead mode, the system generates a checkpoint, which can be used to return execution to the instruction that caused the system to enter the execute-ahead mode. The system then executes subsequent instructions in the execute-ahead mode.

[0012] In a further variation, if the launch point stall condition (the unresolved data dependency or the non-data-dependent stall condition that originally caused the execute-ahead processor to exit the normal execution mode) is finally resolved, the system uses the checkpoint to resume execution in the

normal execution mode from the launch point instruction (the instruction that originally encountered the launch point stall condition).

[0013] In a variation on this embodiment, executing deferred instructions in the deferred mode involves; issuing deferred instructions for execution in program order; deferring execution of deferred instructions that still cannot be executed because of unresolved data dependencies; and executing other deferred instructions that are able to be executed in program order.

[0014] In a further variation, if all deferred instructions are executed in the deferred mode, the system returns to a normal execution mode to resume normal program execution from the point where the execute-ahead mode left off.

[0015] In a further variation, if some deferred instructions are deferred again, the system returns to the execute-ahead mode at the point where execute-ahead mode left off.

[0016] In a variation on this embodiment, the unresolved data dependency can include; a use of an operand that has not returned from a preceding load miss; a use of an operand that has not returned from a preceding translation lookaside buffer (TLB) miss; a use of an operand that has not returned from a preceding full or partial read-after-write (RAW) from store buffer operation; and a use of an operand that depends on another operand that is subject to an unresolved data dependency.

[0017] In a variation on this embodiment, the non-data-dependent stall condition can include, a memory barrier operation, a load buffer full condition, or a store buffer full condition.

BRIEF DESCRIPTION OF THE FIGURES

[0018] FIG. 1A illustrates a processor in accordance with an embodiment of the present invention.

[0019] FIG. 1B illustrates a register file in accordance with an embodiment of the present invention.

[0020] FIG. 2 presents a state diagram which includes the execute-ahead mode in accordance with an embodiment of the present invention.

[0021] FIG. 3 presents a flow chart illustrating the process of determining whether to return to execute-ahead mode or to enter non-aggressive mode.

DETAILED DESCRIPTION

[0022] The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art, and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

Execute-Ahead Processor

[0023] FIG. 1A illustrates the design of an execute-ahead processor 100 in accordance with an embodiment of the present invention. Execute-ahead processor 100 can generally include any type of processor, including, but not limited to, a microprocessor, a mainframe computer, a digital signal processor, a personal organizer, a device controller and a computational engine within an appliance. As is illustrated in FIG. 1, execute-ahead processor 100 includes instruction cache 102, fetch unit 104, decode unit 106, instruction buffer 108, deferred buffer 112, grouping logic 110, memory pipe 122, memory 124,

arithmetic logic unit (ALU) 114, ALU 116, branch pipe 118 and floating point unit 120.

[0024] During operation, fetch unit 104 retrieves instructions to be executed from instruction cache 102, and feeds these instructions into decode unit 106. Decode unit 106 forwards the instructions to be executed into instruction buffer 108, which is organized as a FIFO buffer. Instruction buffer 108 feeds instructions in program order into grouping logic 110, which groups instructions together and sends them to execution units, including memory pipe 122 (for accessing memory 124), ALU 114, ALU 116, branch pipe 118 (which resolves conditional branch computations), and floating point unit 120.

[0025] If an instruction cannot be executed due to an unresolved data dependency, such as an operand that has not returned from a load operation, the system defers execution of the instruction and moves the instruction into deferred buffer 112. Note that like instruction buffer 108, deferred buffer 112 is also organized as a FIFO buffer.

[0026] When the data dependency is eventually resolved, instructions from deferred buffer 112 are executed in program order with respect to other deferred instructions, but not with respect to other previously executed non-deferred instructions. This process is described in more detail below with reference to FIG. 2.

Keeping Track of Dependencies

[0027] The present invention keeps track of data dependencies in order to determine if an instruction is subject to an unresolved data dependency. In one embodiment of the present invention, this can involve maintaining state information for each register, which indicates whether or not a value in the register depends on an unresolved data dependency.

[0028] For example, FIG. 1B illustrates a register file 130 in accordance with an embodiment of the present invention. Each register in register file 130 is associated with a "not-there" bit, which keeps track of whether a valid operand value is contained in the register, or if the operand cannot be produced because of an unresolved data dependency. If the register is waiting for an operand to return (for example, from a load operation), the corresponding not-there bit is set to indicate that the desired operand value is not present in the register. When a subsequent instruction references a source operand value that is marked as not-there, and generates a result that is stored in a destination register, the system marks the destination register as not-there to indicate that the value in the destination register also depends on the unresolved data-dependency. This can be accomplished by marking the not-there bit of the destination register with the "OR" of the not-there bits for source registers of the instruction.

State Diagram

[0029] FIG. 2 presents a state diagram which includes the execute-ahead mode in accordance with an embodiment of the present invention. The system starts in normal execution mode 202, wherein instructions are executed in program order as they are issued from instruction buffer 108 (see FIG. 1).

[0030] Next, if an unresolved data dependency arises during execution of an instruction, the system moves to execute-ahead mode 204. An unresolved data dependency can include: a use of an operand that has not returned from a preceding load miss; a use of an operand that has not returned from a preceding translation lookaside buffer (TLB) miss; a use of an operand that has not returned from a preceding full or partial read-after-write (RAW) from store buffer operation; and a use of an operand that depends on another operand that is subject to an unresolved data dependency.

[0031] While moving to execute-ahead mode 204, the system performs a checkpointing operation to generate a checkpoint that can be used, if necessary, to return execution of the process to the point where the unresolved data dependency was encountered; this point is referred to as the "launch point." (Note that generating the checkpoint can involve saving the precise architectural state of the processor to facilitate subsequent recovery from exceptions that arise during execute-ahead mode or deferred mode.) The system also "defers" execution of the instruction that encountered the unresolved data dependency, and stores the deferred instruction in deferred buffer 112.

[0032] Within execute-ahead mode 204, the system continues to execute instructions in program order as they are received from instruction buffer 108, and any instructions that cannot execute because of an unresolved data dependency are stored in deferred buffer 112.

[0033] If a non-data-dependent stall condition arises while the system is in normal execution mode 202 or in execute-ahead mode 204, the system moves into scout mode 208. (This non-data-dependent stall condition can include: a memory barrier operation; a load buffer full condition; a store buffer full condition, or a deferred buffer full condition.) In scout mode 208, instructions are speculatively executed to prefetch future loads, but results are not committed to the architectural state of the processor.

[0034] Scout mode is described in more detail in U.S. Patent No. 6,415,356, entitled "Method and Apparatus for Using an Assist Processor to Pre-Fetch Data Values for a Primary Processor," by inventors Shailender Chaudhry and Marc Tremblay. It is also described in U. S. Provisional Application No. 60/436,539, entitled, "Generating Prefetches by Speculatively Executing Code Through Hardware Scout Threading," by inventors Shailender Chaudhry and Marc Tremblay (filed 24 December 2002). It is additionally described in U. S.

Provisional Application No. 60/436,492, entitled, "Performing Hardware Scout Threading in a System that Supports Simultaneous Multithreading," by inventors Shailender Chaudhry and Marc Tremblay (filed 24 December 2002). The above listed references are hereby incorporated by reference to provide details on how scout mode operates.

[0035] Unfortunately, because results are not committed to the architectural state of the processor, computational operations performed during scout mode need to be recomputed again, which can require a large amount of computational work.

[0036] When the original "launch point" stall condition is finally resolved, the system moves back into normal execution mode 202, and, in doing so, uses the previously generated checkpoint to resume execution from the launch point instruction (the instruction that initially encountered the stall condition).

[0037] Note that the launch point stall condition is the stall condition that originally caused the system to move out of normal execution mode 202. For example, the launch point stall condition can be a data-dependent stall condition that caused the system to move from normal execution mode 202 to execute-ahead mode 204, before moving to scout mode 208. Alternatively, the launch point stall condition can be a non-data-dependent stall condition that caused the system to move directly from normal execution mode 202 to scout mode 208.

[0038] When the system is in execute-ahead mode 204, if an unresolved data dependency is finally resolved, the system moves into deferred mode 206, wherein instructions are executed in program order from deferred buffer 112. During deferred mode 206, the system attempts to execute deferred instructions from deferred buffer 112. Note that the system attempts to execute these instructions in program order with respect to other deferred instructions in deferred buffer 112, but not with respect to other previously executed non-

deferred instructions (and not with respect to deferred instructions executed in previous passes through deferred buffer 112). During this process, the system defers execution of deferred instructions that still cannot be executed because of unresolved data dependencies and places these again-deferred instruction back into deferred buffer 112. The system executes the other instruction that *can* be executed in program order with respect to each other.

[0039] After the system completes a pass through deferred buffer 112, if deferred buffer 112 is empty, the system moves back into normal execution mode 202. This may involve committing changes made during execute-ahead mode 204 and deferred mode 206 to the architectural state of the processor, if such changes have not been already committed. It may also involve discarding the checkpoint generated when the system moved into execute-ahead mode 204.

[0040] On the other hand, if deferred buffer 112 is not empty after the system completes a pass through deferred buffer 112, the system can perform a number of actions, which are illustrated in more detail in FIG. 3. The system first determines whether or not to resume execution in execute-ahead mode (step 304). If it determines to do so, the system resumes execution in execute-ahead mode from the point where the execute-ahead mode 204 left off (step 308). Otherwise, the system resumes execution in non-aggressive mode (step 310).

[0041] Resuming execution in non-aggressive mode can involve a number of different options. (1) In one embodiment of the present invention, resuming execution in non-aggressive mode involves returning to execute-ahead mode 204 to resume execution from the point where the execute-ahead mode 204 previously left off (see arc labeled "OPTION A" from deferred mode 206 to execute-ahead mode 204 in FIG. 2). However, if a non-data-dependent stall condition is encountered, instead of entering scout mode 208 the system waits for the non-

data-dependent stall condition to be resolved (see loop labeled "OPTION A," which emanates from execute-ahead mode 204 in FIG. 2).

[0042] Note that if a large amount of computational work has been accomplished during execute-ahead mode, it may be wasteful to throw all of this work away by entering scout mode 208. Recall that in scout mode instructions are speculatively executed to prefetch future loads, but results are not committed to the architectural state of the execute-ahead processor. Hence, at the end of scout mode, the system has to resume execution from the last checkpoint, which in this case is the launch point where the execute-ahead mode was initially entered. Backing up to the last checkpoint discards all of the computational work that was accomplished during execute-ahead mode. Instead of discarding all of this work it may be preferable to not enter scout mode when a non-data-dependent stall condition is encountered, but to instead wait for the non-data-dependent stall condition to be resolved (or for a data-dependent stall condition to be resolved which causes the system to return the system to deferred mode).

[0043] In another embodiment of the present invention, in non-aggressive mode, the system does not return to execute-ahead mode, but instead remains in deferred mode 206, and waits until deferred buffer 112 empties before returning the normal execution mode 202 (see loop labeled "OPTION B," which emanates from deferred mode 206 in FIG. 2). Unlike OPTION A for non-aggressive mode, OPTION B does not accomplish additional work by returning to execute-ahead mode. However, OPTION B is simpler to implement, and it does not allow additional deferred instructions to be generated, which may further delay return to normal execution mode 202.

Determining Whether to Execute in Non-Aggressive Mode

[0044] In order to determine whether to resume execution in execute-ahead mode 204 (in step 304 in FIG. 3), the system can perform a number of different actions. The system can keep track of the number of instructions that have been executed in execute-ahead mode 204. If the number of instructions is greater than a threshold value, *T*, a significant amount of work has been accomplished and the system can enter non-aggressive mode to preserve the results of this work. Otherwise, if the number of instructions is less than *T*, the system can return to regular execute-ahead mode.

[0045] If during non-aggressive mode, the system is forced to wait for a significant number of cycles it may be beneficial to increase the threshold, *T*. For example, *T* can be increased by one instruction for every ten cycles of waiting. Or, alternatively, *T* can be increased by a factor of 1.1 for every 100 cycles of waiting.

[0046] If we decide not to enter non-aggressive mode and instead return to regular execute-ahead mode 204, and if a subsequent non-data-dependent stall condition causes the system to enter scout mode, we can adjust *T* upwards or downwards based upon the number of load misses encountered during scout mode. If very few load misses are encountered, then entering scout mode was not effective at prefetching data values, and the threshold *T* can be increased, so that the system is less likely to enter scout mode. On the other hand, if many load misses are encountered, scout mode was effective at prefetching data values, and the threshold *T* can be decreased so the system is more likely to enter scout mode.

[0047] Also, the value of *T* can self-increment or self-decrement in the absence of any feedback. This is useful if only one side of a reward-punishment scheme is easy to implement.

[0048] Other adjustment heuristics are possible. For example, the system can keep track of the threshold values for the last few decisions, and can store a table of threshold values indexed by bits of the program counter, much like commonly used branch prediction mechanisms.

[0049] The foregoing descriptions of embodiments of the present invention have been presented for purposes of illustration and description only. They are not intended to be exhaustive or to limit the present invention to the forms disclosed. Accordingly, many modifications and variations will be apparent to practitioners skilled in the art. Additionally, the above disclosure is not intended to limit the present invention. The scope of the present invention is defined by the appended claims.